Tejas-CNN: A CNN Accelerator Simulator

A thesis submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

Computer Science and Engineering

by

Aditya Jain 2016CS10335

Under the guidance of

Prof. Smruti R. Sarangi

**Department of Computer Science and Engineering,
Indian Institute of Technology Delhi.**

August 2020.

Certificate

This is to certify that the thesis titled **Tejas-CNN: A CNN Accelerator Simulator** being submitted by **Aditya Jain** for the award of **Bachelor of Technology** in **Computer Science and Engineering** is a record of bona fide work carried out by him under my guidance and supervision at the **Department of Computer Science and Engineering**. The work presented in this thesis has not been submitted elsewhere either in part or full, for the award of any other degree or diploma.

Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Abstract

Convolutional neural networks have made revolutionary advances in areas such as computer vision. However, they are compute intensive so to compute them more efficiently dedicated accelerators are being designed for them. In this thesis, a new JAVA-based CNN systolic accelerator architectural simulator, Tejas-CNN, is presented. Tejas-CNN is able to simulate end-to-end convolutional networks on an arbitrary systolic array-based accelerator. It can simulate the convolution layers in output, weight, and input stationary dataflows with different tiled mapping constraints. Unlike other popular simulators, Tejas-CNN performs real computation on the input image values. It also simulates the memory component of the accelerator for more detailed insights.

Contents

Chapter 1

Introduction

With the recent advances in machine learning, many computer vision tasks such as object detection, object segmentation, object tracking, etc. which were earlier thought to be much difficult have now been showing promising results. One of the major factors for this success is the Convolutional neural networks[1] first proposed by Prof. Yann LeCun. Convolutional neural networks belong to a broader method of machine learning called the artificial neural network. The convolution operation includes the dot product of a filter with an overlapping area of the input and adding the results to produce one value in the output, which is a MAC operation. CNN layers show great results when used with multiple filters and multiple channels of input as shown in [1],[2] and [3]. A typical Convolutional layer is shown in Figure 1.1. The figure shows a layer with single input image of size CxHxW where C is the number of channels in the image, H is the height and W is the width of the image, it computes the output for K filters each of size CxRxS where R and S are the filter height and width respectively. In most of the models today they also compute for any number of inputs together which further increases the computation cost of the layer. The complete convolution layer can be described as nested loops shown in listing 1.1 where O, I and W are the output, input, and weights respectively:

Listing 1.1: 2D Convolution in nested loops

```
for (n=0; n < N; n++) {
 for (k=0; k < K; k++) {
  for (c=0; c < C; c++) {
   for (q=0; q < Q; q++) {
    for (p=0; p < P; p++) {
     for (r=0; r < R; r++) {
      for (s=0; s < S; s++) {
        O[n][k][q][p] += I[n][c][q+r][p+s]
                           * W[k][c][r][s];
}}}}}}}
```

Although today most of the traditional vision methods are being outperformed by these CNNs based methods, the computing power is significantly high mostly because of these convolutional layers. Many specialized GPUs are used to train these networks and therefore a lot of research is going on to create low power and dedicated microchips called CNN accelerators for these kinds of networks. However, it is a long process to check the efficiency of different accelerator designs and the current lack of such a simulator inspires us to create one. In this work, we create a simulator that can correctly determine the efficiency of the chip and help researchers explore different accelerator architecture without actually implementing them on FPGA or ASIC. By simulating on a common platform it can compare the accelerator architectures irrespective of the implementation process. By doing real computation it also provides better ways to find data compression algorithms and evaluate reduction techniques to take advantage of the sparsity of data in the CNNs.

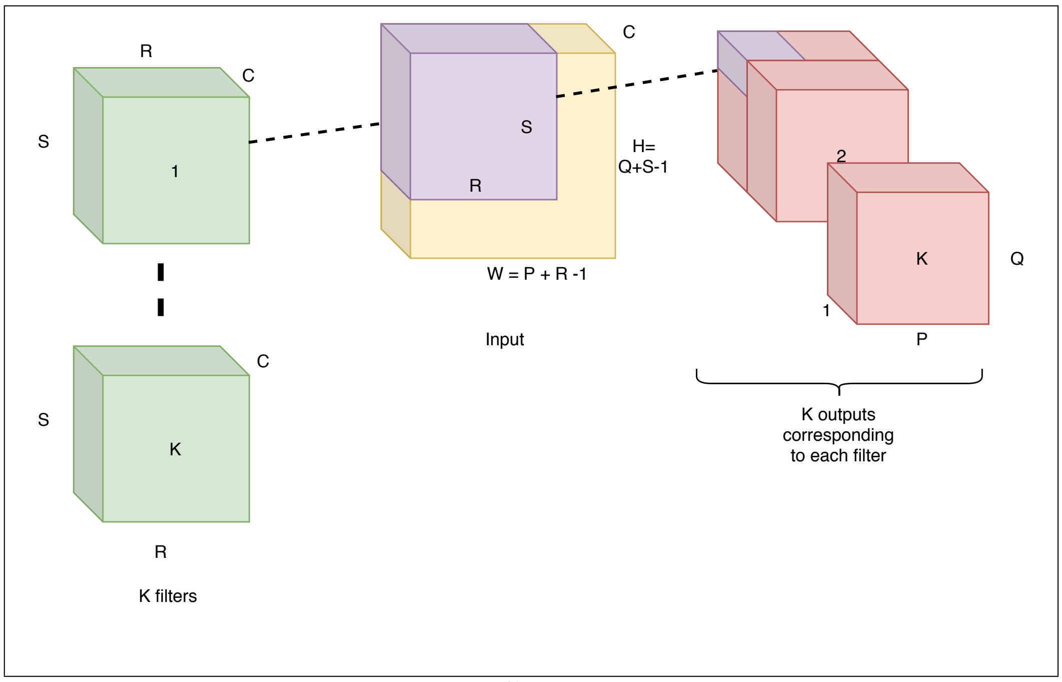

Figure 1.1: Convolution Layer

Chapter 2

Background

2.1 Need for the Accelerator

Deep Neural Networks have found their application in a wide variety of tasks such as image recognition, speech synthesis, etc. Some of these architectures are so dense that they require about 30 billion computations to process a single image. Apart from such a large number of computations, many DNNs also require large amounts of memory to store DNN parameters learned and to store the intermediate results. Currently, GPUs are being used to these kinds of operations, but with the growth in popularity of DNNs, much denser architectures will be proposed to perform a variety of tasks and to do this at faster speeds and in low power environments. This calls for the need for specialized architectures which are set-up to reduce the number of computations by removing the redundancies in the data, and this is where specialized CNN Accelerators come into play. GPUs are Graphical Processing Units which are specialized hardware for manipulation of images and calculation of local image properties. As most of the image calculations are matrix-based, it has led to widespread usage of GPUs even for Convolutional Neural Networks, but there are certain specific advantages of using an accelerator over a GPU for such purposes like it uses low power and the computations in an accelerator can be adjusted according to our precision levels and a more optimized and efficient memory model can be implemented specifically to the architecture which we want to run on the accelerator.

2.2 The Accelerator Designs

Different types of architectures have been explored for the accelerators. These include tightly-coupled 2D systolic-arrays based accelerators such as Google's

TPU[4] and ARM's Project Trillium. Other approaches are based on 1D SIMD based accelerators such as Microsoft Brainwave. While are they both have some pros and cons, systolic arrays offer greater computer density as data is entered from specific points so individual PE's do not require special instructions but on the other hand SIMD has dedicated instructions for data delivery. Empirically it has been shown that TPUs have greater TOP/mm^2 compared to GPUs. Systolic arrays are further easier to operate because of limited possible movement of the data however this comes at the cost of flexibility. Section 2.2.1 explains Systolic Arrays in greater detail.

2.2.1 Systolic Arrays

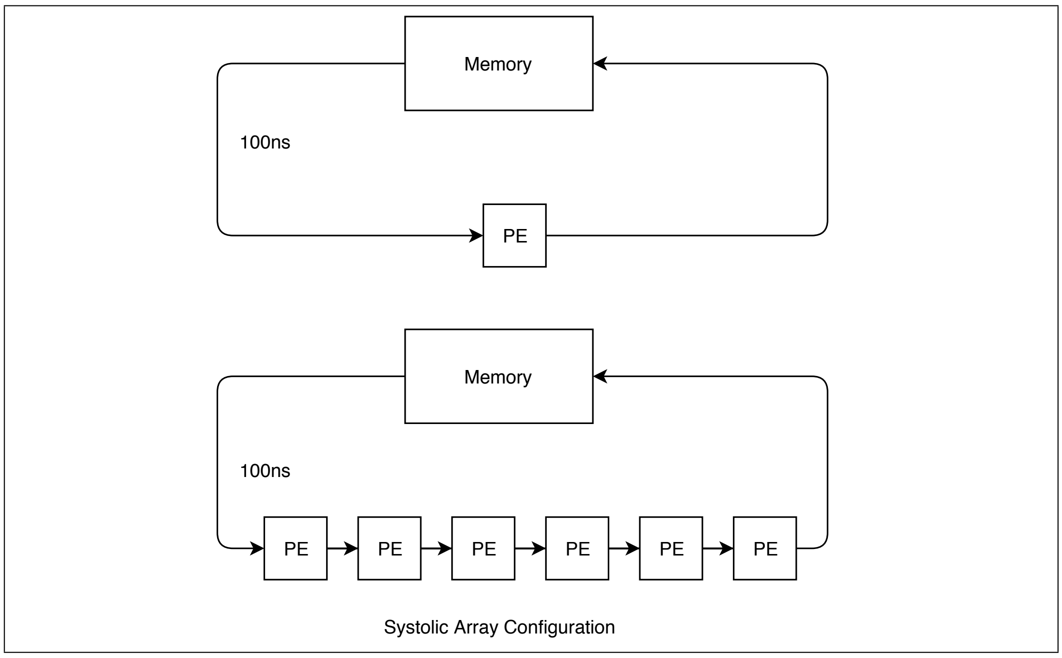

Figure 2.1: Basic structure of Systolic as proposed in [5]

Systolic arrays were first introduced in [4]. They were proposed for parallel computation of linear algebra and matrix-based tasks. A systolic array consists of a monolithic network of programming engines (PE) which can either be hardwired or software configured for specific applications. Data is fed in from specific nodes and all the internal programming engines receive data from their interconnected neighbor PEs. Each PE can also store the intermediate result and pass it to its neighbor. Generally, PEs perform very limited and specific types of operations, as for accelerators most of them can perform addition and multiplication. Since they store the partial data inside the processor array, they do not need to access the external buses, main memory, or external caches during each operation providing significant speedup. Systolic arrays, therefore, have been widely used in tasks such as image processing, pattern recognition, and computer vision. For the Figure 2.1, suppose the time it takes the data to reach from memory to the PE is 100ns then due to the IO limitation the maximum number of operations which can be performed is capped at 5 million operations per second (MOPS) in the first case, while if we use the systolic array-based architecture we can get 30 MOPS since the time for transfer of data from one PE to another is very small as compared to the transfer from memory to PE. This is helpful if the operations which are being performed either depend on one another or you want to compute different operations on the same data. Convolution operation which reuses data are able to exploit this property of systolic arrays.

2.2.2 Systolic Array based Accelerators

Systolic array-based accelerators are becoming more popular these days. Most of them contain a 2D Systolic array instead of a 1D Systolic array. One of the first accelerator designs proposed was of NeuFlow[6], it has a 2D grid of programming tiles along with smart DMA, a runtime configurable bus, and a controller. The current bottleneck inefficiency is the data transfer. Since convolution operation is simply multiplying and addition operation, specific programming engines are designed which can perform these operations on the go. The convolution operation uses the same data many times for the computation so data transfer becomes the bottleneck. Different architectures try to use them efficiently by different parallelism methods. CNP[7] uses the specific hardwired programming tiles and have all the convnet operations provided as macro-instructions. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network[8] tries to reduce memory bandwidth by using short fixed-point numbers instead of using long-floating point numbers. Memory Access Optimized Routing Scheme for Deep Networks on a Mobile Coprocessor[9] tries to reduce the redundant data requests by using the data for multiple outputs at once and storing the intermediate results. A Dynamically Configurable Coprocessor for Convolutional Neural Networks[10] exploits the two types of parallelism found: Intra (one output, multiple inputs) and inter (one input, multiple outputs). By allowing arbitrary control over the extent of each parallelism it is able to achieve great performance. Figure 2.2 shows the architecture of an accelerator containing

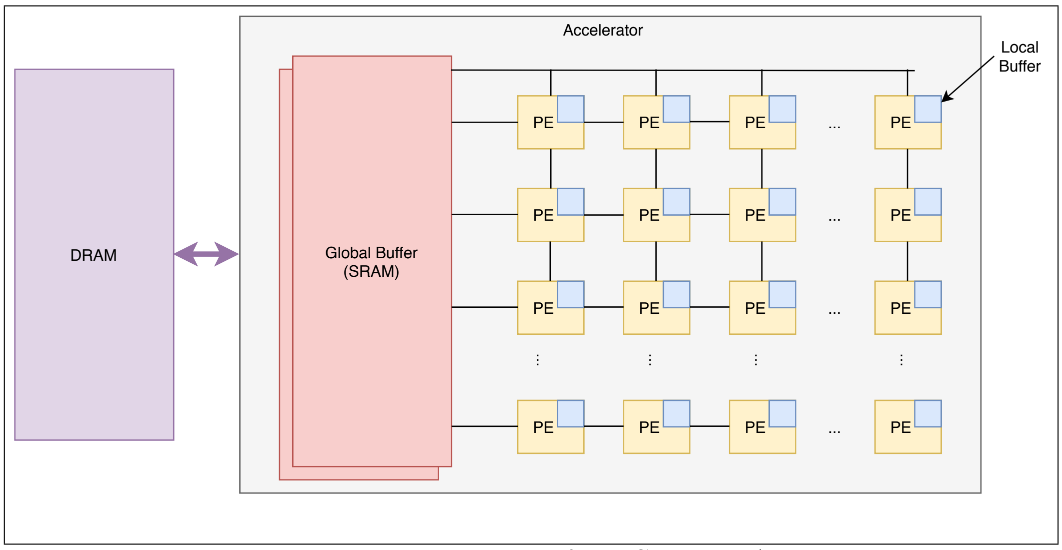

Figure 2.2: Basic structure of 2D Systolic Accelerator

a 2D systolic array of programming engines (PE). Each PE has a local buffer similar to an L1 cache and the accelerator has a large SRAM acting as a global buffer(GB) used to store inputs, weights, and outputs. The complete data is stored in an off-chip DRAM. The global buffer is doubly buffered so that PE array does not need to wait for the data to come from DRAM. The PEs are interconnected with an NoC. Each PE can communicate with its adjacent PEs in systolic fashion. There are multiple ways in which we can tile, reorder, and parallelize the loops of the convolution operation.

2.3 Mapping Scheme

A mapping scheme determines how the data is split in tiles at each memory level and how it is spatially distributed among the multiple instances of PEs in the array for parallelization. We follow a *loop nest* based mapping as described in Timeloop[11]. There are three levels of memory: DRAM, GB, and local buffer of PE so we can break one *for* loop for one index variable from the 7 nested loops of convolution into three nested loops, where index variables are *P,Q,R,S,C,K and N*. Furthermore, we can parallelize for multiple PEs which can be written as an additional *for* loop whose iterations are spatially distributed among the PEs. Overall the mapping can be represented as:

Listing 2.1: Mapping in nested loop format

```
// DRAM
for(p2=0;p2<P2;p2++)
  ..
  for(n2=0;n2<N2;n2++)

  // Global Buffer
  for(p1=0;p1<P1;p1++)
    ..
    for(n1=0;n1<N1;n1++)

    // Spatial GP->PE
    // Parallel for loops
    for(s_p0=0;s_p0<S_P0;s_p0++)
      ..
      for(s_n0=0;s_n0<S_N0;s_n0++)

      // PE/Local buffer
      for(p0=0;p0<P0;p0++)
        ..
        for(n0=0;n0<N0;n0++)
          p = p2*P1*S_P0*P0 + p1*S_P0*P0
                +s_p0*P0 + p0;
          ..
          n = n2*N1*S_N0*N0 + n1*S_N0*N0
                +s_n0*N0 + n0;
        O[n][k][p][q]+=I[n][c][p+r][q+s]
                  * W[k][c][r][s];
```

Note the following from the listing 2.1 :

- For any index variable the multiplication of tiling limits at each level should be equal to the original index variable value.
- At each level the factor values for each index variable and the permutation of the loops can define the mapping at that level.

2.4 Dataflow for Systolic Array based Accelerators

Dataflow plays a crucial role in the performance of accelerators. There are many dataflow methods that can be used, we focus on the three most common ones namely: Output Stationery, Weight stationery, and Input stationery. The stationary term comes from the fact that data is cached at the local buffer of the PE and based on which type of data is cached for reuse we define the stationary types. In output stationery, each programming engine is given the task to compute one output value. While in the case of weight stationery the filter weights are pre-filled in the programming engines and do not move. Similarly in input stationery instead of filter weights, the input values are pre-filled in the programming engine nodes. All three dataflow's performance depends on the accelerator design and the neural network which is being run, so there is no one optimal dataflow for every situation.

2.5 Simulators

As accelerators are becoming more popular, the need for good simulators is also increasing. Recently a new tool, Accelergy[12], was released which simulates energy consumption. Also, a systolic array-based accelerator simulator Scale-sim[13] by researchers at ARM can simulate convolution and matrix-matrix operations and determine the efficiency. Since it is a systolic accelerator simulator it provides the user with the ability to choose the type of dataflow they want from these three: weight stationery, output stationery, and input stationery. Scale-Sim provides the cycle counts and utilization of the architecture without actually doing the computations. So it can not be used to discover better data compression and reduction methods. Timeloop[11] is another tool for evaluating dnn accelerators. It also tries to find the optimal mapping based on energy and performance metrics without performing real computations.

2.6 Advantages of Simulators

Using a simulator helps in evaluating the performance of the hardware designs without actually physically building them, this helps in lowering the cost and enable faster exploration of the designs. It also helps in simulating hypothetical hardware designs that are not possible or are very hard to realize. Also by allowing to run different neural network models, it can also help in designing such models so that they can utilize the resources efficiently on the provided accelerator thus helping not only the accelerator researchers and designers but also the application developers. Simulators are also able to provide functionalities that normal hardware would not allow such as running the code backward in case an error has been detected to debug it easily.

Chapter 3

High-level Design

Tejas-CNN is a highly modular cnn-accelerator simulator that can simulate any neural network with supported layers from end to end given valid mappings on an input accelerator design even considering memory(GB+DRAM) modules. It performs real computations for any of three stationary types: output stationary, input stationary, and weight stationary and produces cycle accurate metrics.

3.1 User Interface

Figure 3.1 shows the end to end pipeline for using Tejas-CNN. The main simulator takes a config file (XML) as an input from the user which contains the following required data:

- **Accelerator Design:** This defines the shape of the systolic array i.e. the number of rows and columns of the systolic array. Along with this, the stationary type for the dataflow is also specified as an input for simulation. The design also includes the parameters for the DRAM and Global buffer.

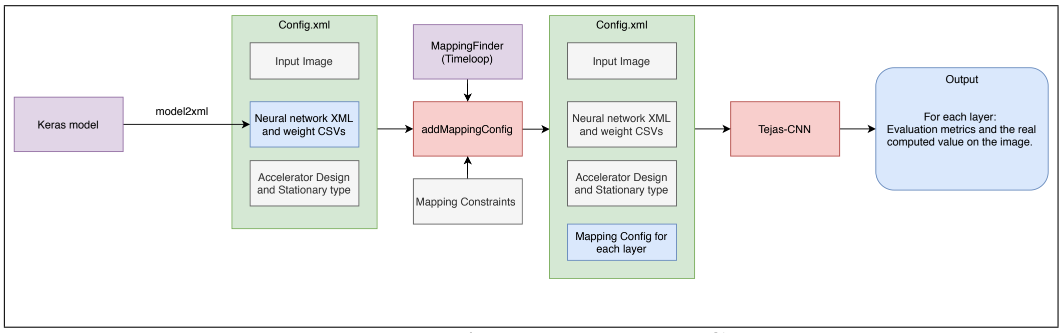
Keras model
model2xml
Config.xml
Input Image
Neural network XML and weight CSVs
Accelerator Design and Stationary type
MappingFinder (Timeloop)
addMappingConfig
Mapping Constraints
Config.xml
Input Image
Neural network XML and weight CSVs
Accelerator Design and Stationary type
Mapping Config for each layer
Tejas-CNN
Output
For each layer: Evaluation metrics and the real computed value on the image.

Figure 3.1: End to end pipeline for running Tejas-CNN with arbitrary keras model and mapping

- **Input Image:** Since Tejas-CNN does real computations, instead of just taking the shape of the input, it takes real pixel values. This can allow us to explore optimization techniques in the future by trading precision or other methods.
- **Neural Network:** This is the end to end neural network that has to be simulated. The network is described in a fixed XML format to unify the representation of the network across all the frameworks. All the parameters of a convolution layer like stride, padding, activation, etc. are defined in this XML file. For convenience Tejas-CNN comes pre-equipped with a script(*model2xml.py*) to convert any keras, which is a Tensorflow[14] based API, based model into the required format.
- **Mapping:** Each convolution layer can be simulated in different ways based on the mapping specified in the format described in Section 2.3. The config file should contain a separate mapping for each of the convolution layer. It is to be noted that Tejas-CNN does not compute the optimal mapping but can run any valid mapping.

Taking these inputs Tejas-CNN produces two files for each layer in a separate *results* folder, one with name *layerName.txt* containing the stats for the simulation defined in Section 5 and other with name *layerNameComputOutput.txt* containing the real output for that layer with the given input image.

3.2 Code Structure

Tejas-CNN's simulation code is entirely written in JAVA. It consists of the following packages:

- **main:** The main package contains the main calling class, *Tejas-CNN*, of the simulator which is responsible for loading the input image and neural network data and calls functions to simulate layers sequentially with the corresponding input.

- **accelerator:** This package is used to model the accelerator for simulation. It contains support for modular PE structure by having separate ALU and register file classes. It also contains the memory modules for DRAM and global buffer. It's DRAM module is borrowed from Tejas[15] and changed according to needs. The inter PE memory and the systolic array structure of PEs is also defined in this package.
- **config:** This package contains classes to store the configuration of the main memory, the global buffer memory, and the mappings for different layers. The mapping config is unique for each convolution layer based on its input dimensions.
- **input:** It contains the support for loading the data into the simulator and its classes are called by the main class once. The XML parser for the config file and input loader for the weights from csv file and input images are all part of this package.
- **network:** This package is used to create neural network abstraction for the simulator. It contains a Layer class which is the base class for all the layers i.e. the other classes for layers like Conv, Maxpool, etc. are all children of this class. Each class of a layer contains complete information about the parameters for that layer which are loaded from the network's XML file.
- **tiledmapper:** This package maps the data that has to be simulated on the accelerator according to the mapping and stationary type specified in the input. It takes care of the spatial and temporal constraints and prepares the row and col input for the systolic array, both of which are 2D arrays. It also has classes to define the constraint and problem size. This package is mainly created by *Chinmay Rai* except for the oneCycleOperation function.
- **scheduler:** This package contains the main simulation element of the software. It takes input from the tiledmapper and simulates the convolution operation on the architecture specified by the accelerator module. It has separate classes for simulating different stationary types.

- **scripts:** This folder contains some additional scripts which are not part of the simulation but helps in preparing the input data for the simulator.

3.3 Integration of Timeloop

As mentioned earlier, Tejas-CNN does not compute the optimal mapping for either performance or energy consumption but runs all the valid mappings, so in order to simulate better mappings, we use the Timeloop's mapper[11]. Timeloop searches for an optimal mapping based on some problem and architectural constraints. The scripts/addMappingConfig.py file takes three arguments: timeloop's path, the input image, and config XML to run the timeloop's search and write the mappings into the config file of Tejsa-CNN. The input image is required in order to find the problem's constraint for each layer. The default constraints used are based on the stationary type given but custom constraints can also be given for creating mappings.

3.4 Simulating a Convolution layer

Figure 3.2 shows the flow chart for simulating one Conv layer in the program. Since the accelerator is doubly buffered, at a time one global buffer is providing data to the systolic array while the other one is concurrently pre-fetching data from the DRAM for the next block. The buffers are swapped when the chunk of data contained in one iteration of the nested loops at the DRAM level is used otherwise the mapper provides data for the next iteration of the nested loops at the Global buffer level. The flow chart shows the OS stationary convolution to be simulating but the same can be used for IS and WS too with changes in mapping and scheduler only.

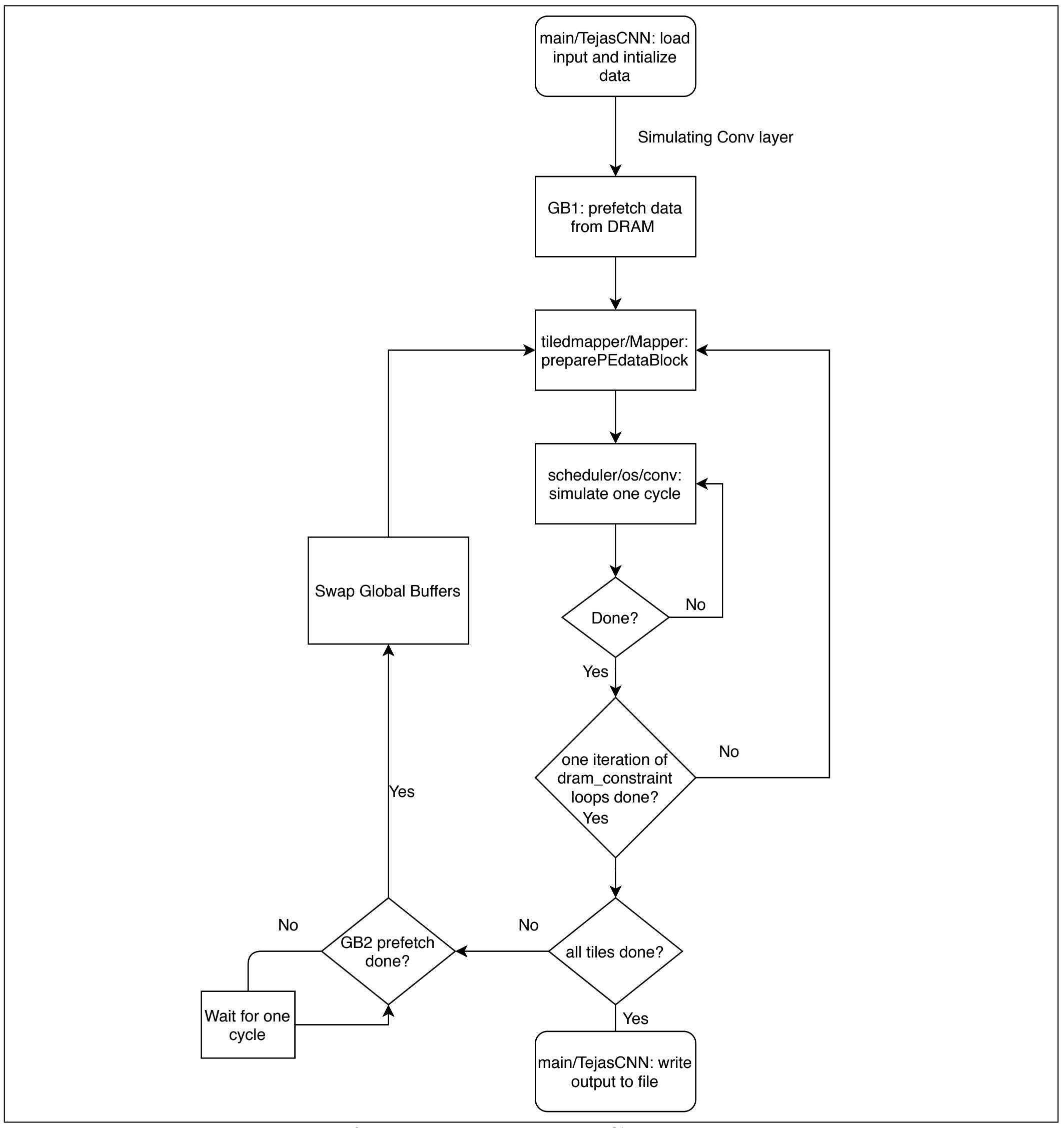
main/TejasCNN: load input and intialize data
Simulating Conv layer
GB1: prefetch data from DRAM
tiledmapper/Mapper: preparePEdataBlock
scheduler/os/conv: simulate one cycle
Done?
No
Yes
one iteration of dram_constraint loops done?
No
Yes
all tiles done?
No
Yes
GB2 prefetch done?
No
Yes
Wait for one cycle
Swap Global Buffers
main/TejasCNN: write output to file

Figure 3.2: Flow chart for simulating one Conv layer in the simulator. Due to double buffering while one buffer is being used by the PEs, the other is pre-fetching the data from the DRAM

Chapter 4

Implementation

Iterative style is used for the simulation which means that in each iteration all the components are advanced through one clock cycle. This methodology is adopted because accelerators are designed in such a way that the systolic array does not stall so in each cycle some PEs are active and are computing and passing the data to its neighbor or back to GB. Furthermore, we pass the real values of the input to the PEs for computation.

4.1 Constraints

To enforce the mapping scheme as described in Section 2.3 in our simulator we define some constraints. First we call the parameters P,Q,R,S,C and K of the convolution operation as problem index variable or simply *index variables* (Idx). Then we define a constraint as in listing 4.1. It contains *type* to denote whether the constraint is for spatial/parallel *for* or not. Furthermore, each constraint object is used to represent the loops for all index variables at one level of memory hierarchy, so to describe it completely as shown in listing 2.1 we need to know the loop order which is stored in the *permutation* variable and the tile size/factor value for each index variable which is stored in the *factor* variable. To describe the complete mapping we need four of these constraint objects: one at the DRAM level, one at the GB level, one for the parallelizing or spatial division of the workload on the PE array, and one at the register level of the PE. We call these constraints *dram_constraint*, *gb_constraint*, *spatial_constraint* and *pe_constraint* respectively from here on.

Listing 4.1: Defining Constraints

```
public class Constraint{
    String type;
    List<Idx> permutation;
    Map<Idx, Integer> factors;
}
```

4.2 Simulating the systolic array

Since the whole convolution operation is divided into multiples tiles, the systolic array computes the convolution operation for each tile size sequentially. From the listing 2.1, it can be seen that the size of one tile which will be computed is defined by the parallel for loops and the loop constraints on the register file of PE. In terms of constraint objects, the tile size is defined by factors of the *spatial_constraint* and the *pe_constraint*. Once we have the input feature map and filters according to the described constraints, it can be mapped to the systolic array in three different dataflows as described below.

4.3 Output Stationary

As the name suggests, in this dataflow the partial sum is stationary at PE level, that is each PE computes the multiply and add operations for one output pixel. The data is passed in the following manner:

- Each column takes care of one filter. The values of that filter are unrolled and passed one by one in each cycle through the array in a staggering fashion.
- The input pixel values in the convolution window for each output pixel are passed horizontally through the systolic array with appropriate staggering similar to the weights.
- After complete operations for an output pixel value it is sent to the GB in systolic fashion that is each PE passes the results to the PE above it and the PEs in the top row sends them back to GB.

Since our simulation method is iterative we advance all the data through the PEs according to the above mentioned scheme for input, weight, and output for each cycle. Figure 4.1 shows an example for output stationary dataflow for convolution of a 4x5x5 size input feature map with 16x4x2x2 filters on a 16x16 systolic array. If the number of filters is more than the number of columns of the PE array or the convolution window size is more than the number of rows then *folding* occurs but the mapping is created in such a way as to avoid that situation.

In terms of constraints, since we want the output to be stationary and the output depends on P, Q, C, and K index variables, the factor values for these index variables in *pe_constraint* should be one. Similarly, since the convolution window's pixels are not divided among the PEs, the factor values for the index variables R, S, and C in the *spatial_constraint* should be 1.

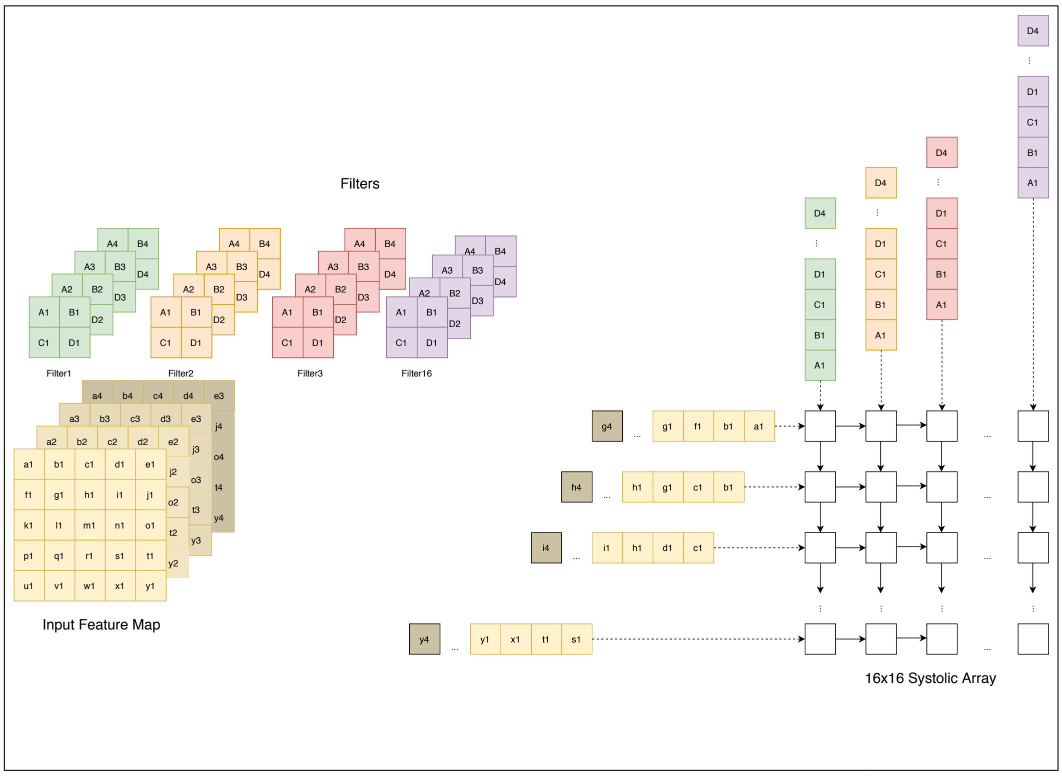

Figure 4.1: Example of output stationary on convolution of 4x5x5 input feature map with 16x4x2x2 filters

4.4 Weight Stationary

Unlike output stationary, the weights are made to reside in PEs in weight stationary. Weight stationary has two phases:

- **Pre-filling Weights**: In this phase, the weights are prefilled in the systolic array entering from the top row. Each column's PEs have filter values stored corresponding to one filter out of K.
- **Input Passing**: After the weights are prefilled, the input feature map values in one convolution window is unrolled and passed through the systolic array. The partial sum generated at PEs is then accumulated in systolic fashion by passing them upwards.

Figure 4.2 shows an example of the input passing phase of the weight stationary. For constraints, since the weights are stationary and they depend on the R, S, C, and K index variables, the factors for these variables in the pe_constraint should be 1.

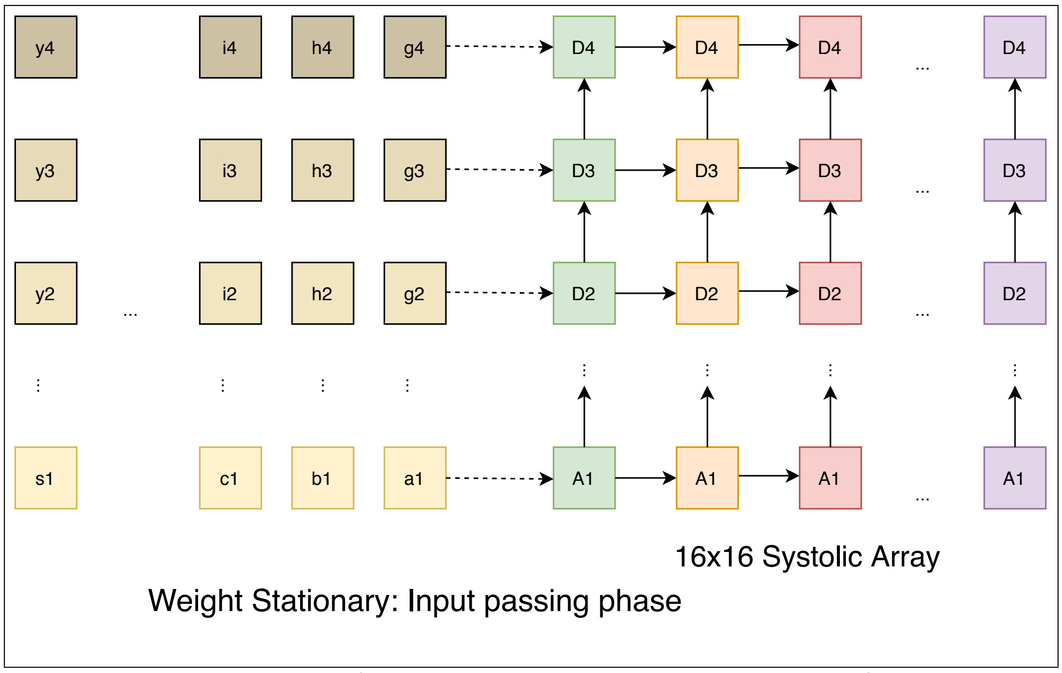

Figure 4.2: Example of weight stationary on convolution of 4x5x5 input feature map with 16x4x2x2 filters

4.5 Input Stationary

Similar to weight stationary input stationary has two phases but the prefilled values are input feature map values instead of weights and weight values are passed through the systolic array instead of input values. Since input feature map values depend on P, Q, and C, the factors for these index variables in pe_constraint should be 1. Furthermore, the factor value of the K index variable in the spatial_constraint should be 1 since it is not parallelized.

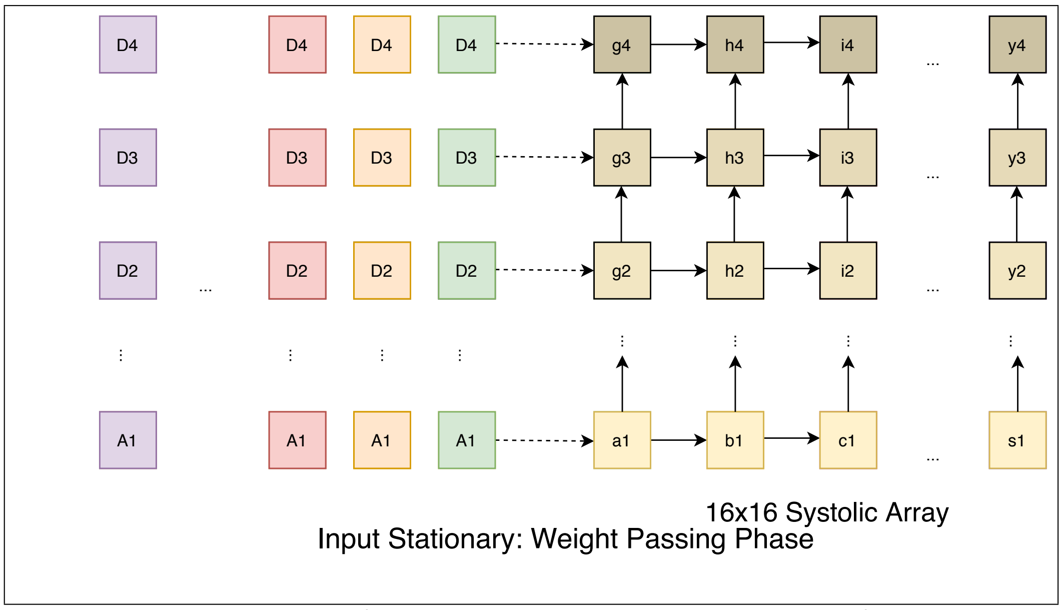

Figure 4.3: Example of input stationary on convolution of 4x5x5 input feature map with 16x4x2x2 filters

4.6 Memory Interaction

Apart from PE register file, Tejas-CNN provides support for both GB and DRAM. For global buffer, unlike cache, the data arrangement is mainly software programmed, so it is assumed that the data is arranged in it in such a manner that no stalling of the systolic array occurs due to it. However, it places the limit on the number of elements (values requested by PEs) accessed in a cycle to the number of banks available in the global buffer and stalls if the limit is exceeded by the elements requested. For DRAM simulation, Tejas-CNN borrows the *MainMemoryController* from Tejas[15] and is modified to integrate with the rest of the simulator. The original component was designed to support multiple cores which are removed since only the simulator will send and receive data from it. The eventQueue is retained but it is used only for the communication between the global buffer and the controller.

4.7 Prefetching and Output Writing

The request for the prefetching is added by the global buffer based on dram_constraint loops, sending requests for inputs and weight used by one iteration of those nested loops. After adding the request in the eventQueue, the controller puts each request in its own pending queue. In each subsequent cycle, the controller calls the enqueueToCommandQ function which takes the next pending request to work on. The controllers oneCycleOperation function is called in each cycle along with the one cycle simulation of the systolic array. When the simulation is complete for one iteration of the nested loops defined by dram_constraint, the output is written back to the controller from the global buffer after sending a write request along with the addresses. Since these addresses depend on the mapping scheme, they are computed before starting the simulation.

The controller along with request type requires the virtual address for the data to be read or written. A unique number,v, for each input feature map pixel value which is located at the x row, y col and c channel is computed as:

$$v = c * (X + Y) + x * Y + y \tag{4.1}$$

where X and Y are the total numbers of rows and cols of the input feature map respectively. The value in v is shifted to account for the size of the pixel value in bytes as the address is byte-addressable and a base address is added to get the virtual address. In a similar way addresses for weight and output pixel values are generated with different base addresses.

Chapter 5

Evaluation and Analytical Model

For each convolution layer after the simulation, Tejas-CNN produces the number of cycles taken, prefilling cycles, the global buffer accesses made, and the dram accesses which can be used to further get energy using available tools. We evaluate three different workloads of convolution layer taken from VGG16[3] as described in Table 5.1.

Name	Input Size (CxHxW)	Filter Size (KxCxRxS)
Conv1	3x128x128	64x3x3x3
Conv2	64x64x64	128x64x3x3
Conv3	128x64x64	128x128x3x3

Table 5.1: Workloads for evaluation

We run these workloads on Tejas-CNN for all three dataflows described in Section 4.2. Since there are multiple valid mappings and a mapping can change the performance and all the other statistics for any dataflow, mappings were chosen for each stationary and workload such that the PE utilization was almost equal giving a better comparison between the stationary types. But it should be noted that based on the mapping Tejas-CNN receives, the results can vary for the same workload. Parallel work is also being done to find the optimal mapping for a workload by *Chinmay Rai* which will be added later.

The total number of cycles depends on the number of tiles and the cycles taken for one tile computation. Table 5.7 shows these values for all dataflows and workloads and Figure 5.1 shows the graph for the total cycles for them. For a deeper insight, the reuse of weight, input, and output values in their respective stationary dataflows for one tile and all the workloads is shown in Table 5.9. The reuse defines for how many MAC operations the data was loaded only once in the systolic array. For example in the case of WS and

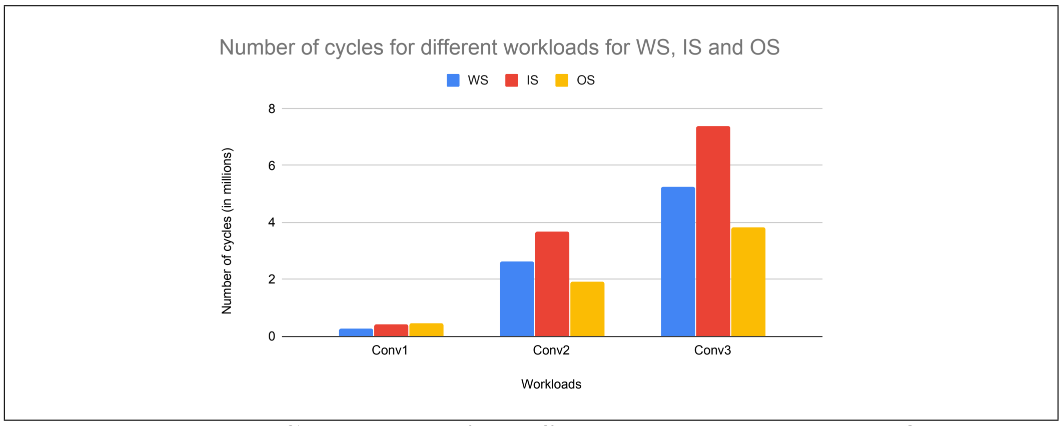

Figure 5.1: Cycle counts for different workloads and dataflows

Conv1 workload, from Table 5.9 the weights loaded once were used for 128 MAC operations, however, the total number of MAC operations they were required in was 16384 (PxQ) which means that they would be accessed 128 times only. Another way to say this is that the number of accesses is reduced by the factor of the reuse value (described in terms of mappings in Table 5.8). This affects the performance too as evident by Figure 5.1 and Table 5.9, it can be seen that higher the reuse value lower the total number of cycles. This deduction can help in designing mappings by using the relation between the reuse and mappings as shown in Table 5.8. Furthermore, Conv2 and Conv3 workloads' tile computation cycles are the same for each dataflow but their total number of cycles is different due to the number of tiles because Conv3 has double the number channels as Conv2 in its problem definition.

Values	**x**	**y**	**t**
Weight stationary	sp.R * sp.S * sp.C	sp.K	pe.P * pe.Q
Input stationary	sp.R * sp.S * sp.C	sp.P * sp.Q	pe.K
Output stationary	sp.P * sp.Q	sp.K	pe.R * pe.S * pe.C

Table 5.2: Values of x,y, and t for different dataflows as shown in Section 5.1, sp.Idx = spatial_constraint.factors[Idx], pe.Idx = pe_constraint.factors[Idx]

The GB accesses for weights are small in case of weight stationary, and a similar pattern is seen in input and output GB accesses for input and output stationary respectively. From Table 5.3, it can be argued that this pattern will follow as long as t is greater than x and y, or to say that reuse of the

	Weight stationary			**Input stationary**			**Output stationary**		
	Inputs	Outputs	Weights	Inputs	Outputs	Weights	Inputs	Outputs	Weights
Global Buffer	x * t	y * t	x * y	x * y	y * t	x * t	x * t	x * y	y * t

Table 5.3: Memory Access Counts for Global Buffer for one tile in terms of x,y and t from Table 5.2, as shown in Section 5.1

	Weight stationary		**Input stationary**		**Output stationary**	
	Pre-filling	Compute	Pre-filling	Compute	Pre-filling	Compute
Cycle counts	y*	t+x+y-1	y*	t+x+y-1	NA	t+2x+y-3
Total	t+x+2y-1		t+x+2y-1		t+2x+y-3	

Table 5.4: Cycle Counts for one tile as formulated in Section 5.1 using Table5.2, *Assuming Pre-filling is done from side

	DRAM Accesses
Input	$(C/dc.C) \times (P/dc.P + R - 1) \times (Q/dc.Q + S - 1)$
Output	$(P/dc.P) \times (Q/dc.Q) \times (K/dc.K)$
Weight	$(R/dc.R) \times (S/dc.S) \times (C/dc.C) \times (K/dc.K)$

Table 5.5: Memory Access Counts for DRAM in one iteration of dram_constraint loops as shown Section5.1.4; P, Q, R, S, C and K are the values in the defined problem; dc.Idx (Idx is the index variable) is used to denote dram_constraint.factors[Idx]

	Weight stationary			**Input stationary**			**Output stationary**		
	Inputs	Outputs	Weights	Inputs	Outputs	Weights	Inputs	Outputs	Weights
Global Buffer	768	1024	192	192	512	384	1152	256	1152
DRAM	792	2048	1152	3168	8192	2304	3168	4096	1152

Table 5.6: Memory Access Counts for Global Buffer and DRAM for one iteration of nested loops of gb_constraint and dram_constraint respectively for Conv2 workload

data should be larger than both number of cols used and the number of rows used. However, DRAM accesses depend only on the dram_constraint factors and therefore such a pattern is not to be expected in its graph as shown in Figure 5.3.

	Weight stationary				**Input stationary**				**Output stationary**			
	Prefill	Comp	Total	#tiles	Prefill	Comp	Total	#tiles	Prefill	Comp	Total	#tiles
Conv1	16	152	168	1536	16	56	72	6144	NA	56	56	8192
Conv2	16	91	107	24576	16	59	75	49152	NA	117	117	16384
Conv3	16	91	107	49512	16	59	75	98304	NA	117	117	32768

Table 5.7: Cycle counts: Prefill, Comp (Compute) and Total columns are the cycle counts for one tile for the dataflows. The #tiles column shows the number of total tiles for the respective workload and dataflow.

Dataflow	Reuse for respective stationary type
Input Stationary	$pe.K$
Weight Stationary	$pe.P \times pe.Q$
Output Stationary	$pe.R \times pe.S \times pe.C$

Table 5.8: Reuse of input, output and weight values in their respective stationary dataflows for one tile in terms of pe_constraint as shown in Section5.1.1 , 5.1.2 and 5.1.3; pe.Idx = pe_constraint.factors[Idx]

	Conv1	Conv2	Conv3
Input Stationary	32	32	32
Weight Stationary	128	64	64
Output Stationary	27	72	72

Table 5.9: Reuse of input, output and weight values in their respective stationary dataflows for one tile for different workloads;

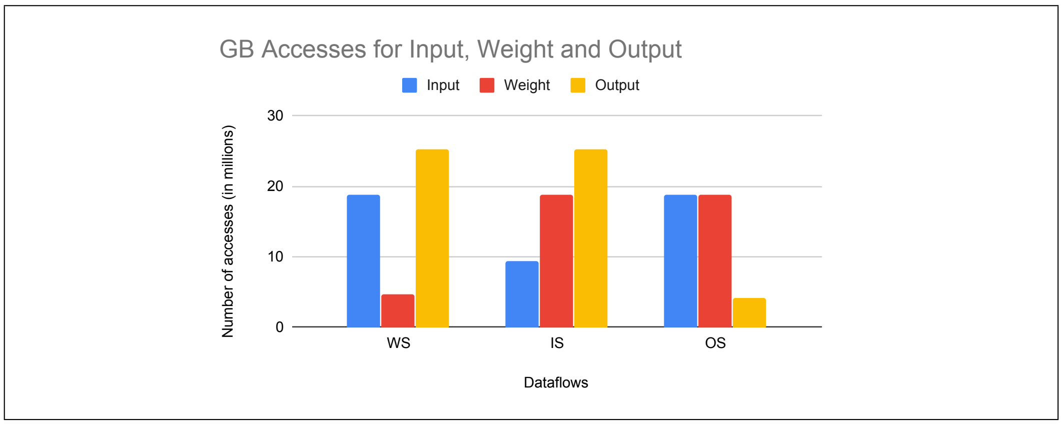

Figure 5.2: GB accesses trend for Conv2 workload for different dataflows

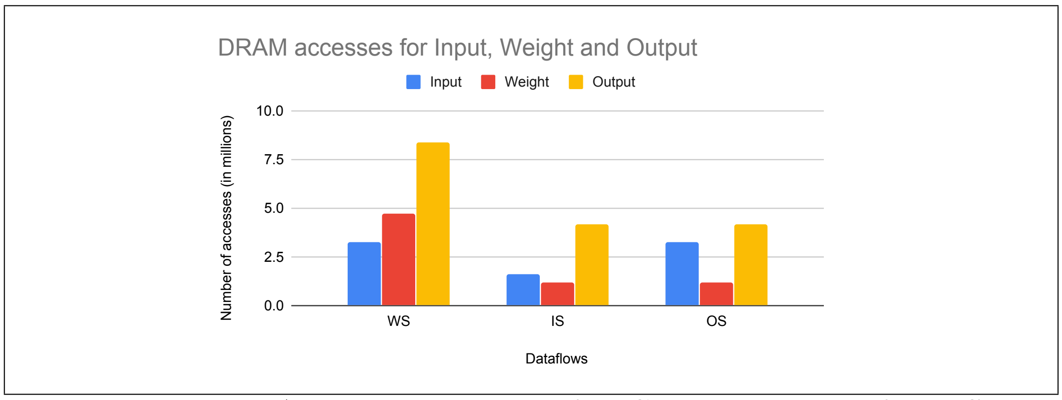

Figure 5.3: DRAM accesses trend for Conv2 workload for different dataflows

5.1 Validation

To show validation of the results an analytical model based on the constraints is formed. This is done due to the unavailability of hardware resources required to run the accelerator. Also, as of now there are no existing simulators which produce cycle accurate statistics for the tiled mappings with which results can be validated, Scale-Sim [13] is cycle accurate but do not consider tiled mappings while Timeloop [11] considers these mappings but is not cycle accurate. We divide the analysis based on the dataflow used. To compute the statistics for the whole operation, first, the statistics for one tile are computed which is then multiplied by the number of such tiles. This is valid since all tiles are of the same size as shown in Section 4.2, and the access and compute pattern remains the same for all tiles irrespective of their order for a fixed dataflow. Furthermore, the mapping is selected so as to fully utilize the systolic array as shown in Figure 4.1, 4.2 and 4.3 but there may be cases when only part of the PEs are active or so as to say the systolic array is underutilized depending on the mapping constraints.

5.1.1 Weight Stationary

For one convolution operation using weight stationary first we define the following :

- *Number of utilized cols (y)*: Since the filters are spatially divided among the PE's as shown in Section4.4, it is equal to *spatial_constraint.factors['K']*.
- *Number of utilized rows (x):*: The values of a convolution windows are placed in rows so this is equal to spatial_constraint.factors['R']* spatial_constraint.factors['S'] *spatial_constraint.factors['C'].
- *Number of unrolled convolution windows (t)*: These unrolled windows are passed through the systolic arrays. One unrolled window corresponds to one output pixel for a filter and since every active PE interacts with values of one unrolled window, the total number of interactions are equal to

  pe_constraint.factors['P']*pe_constraint.factors['Q'].

After defining the above values, we can count the number of cycles required to complete the whole operation assuming no stalls. Counting the cycles for each phase:

- *Prefilling the weights:* If the weights are prefilled from top then the number of cycles in doing so will be x or the number of utilized rows, and if they are prefilled from left to right then the number of cycles will be y or the number of utilized cols. This is assuming that there is no stalling and folding is not required because mapping avoids it.
- *Passing the inputs:* To count the total number of cycles required in this phase break it down into three steps as shown in Figure 5.4:
  - Step 1: The number of cycles required for the last unrolled window to reach the systolic array. This is equal to t or the number of unrolled convolution windows since in each cycle one unrolled convolution window moves horizontally.
  - Step 2: The number of cycles required for the last unrolled window to pass the systolic array after the above step. This is equal to y or the number of utilized cols.

– Step 3: The number of cycles required for the last partial sum generated by the last unrolled convolution window and the right-most filter to reach the topmost PE to write back to GB. This is equal to x or the number of utilized rows.

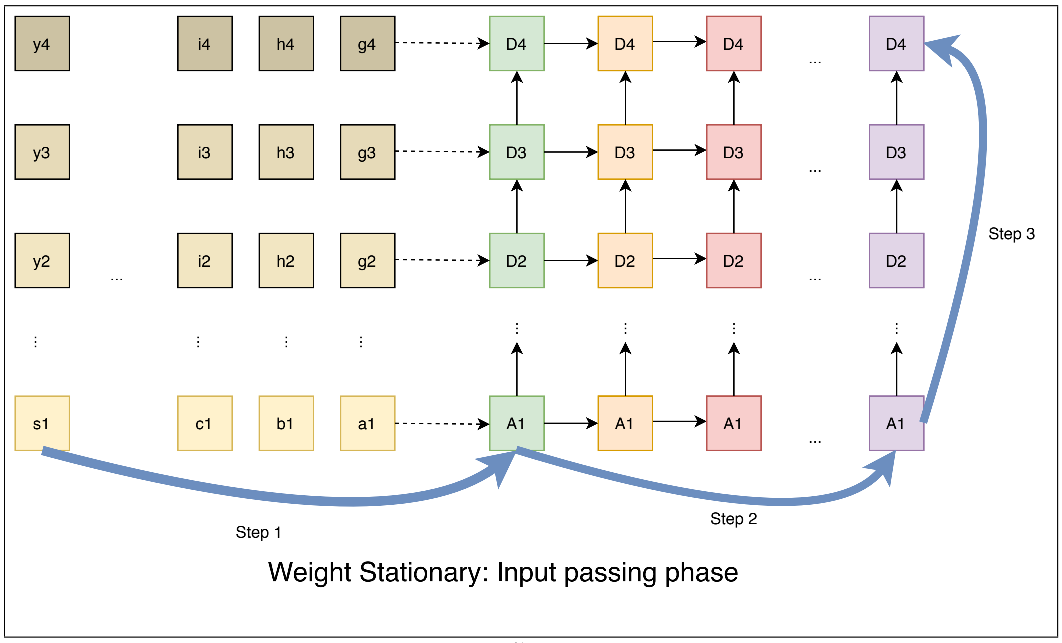

Figure 5.4: Weight Stationary input passing

So the total number of cycles for compute are t+x+y-1 considering overlaps between the steps and the total number of cycles for a tile should be *t+2x+y-1* or *t+x+2y-1* depending on the prefilling. GB accesses for a tile can be broken down into the number of accesses for weights, inputs, and outputs. There are $x \times y, x \times t$ and $y \times t$ accesses for weights, inputs, and outputs respectively. Weights are accessed $x \times y$ times to fill the systolic array, x inputs are accessed for t times while passing the data and for each column, there are t outputs, one from each input unrolled convolution window. Furthermore, the reuse of weights is equal to the number of inputs that are passed through it which is t or $pe.P \times pe.Q$ where pe.Idx represents pe_constraint.factors[Idx].

5.1.2 Input Stationary

Input stationary has similar analysis as weight stationary however x,y and t have different relations with the constraint as described below:

- Number of utilized cols (y): Just opposite to weight stationary, the inputs values in the convolution windows for one output pixel are stored in the PE registers instead of passing. So the number of such windows will be equal to the number of output pixels. This is denoted by the spatial_constraint.factors['P']* spatial_constraint.factors['Q'].
- Number of utilized rows (x): Similar to weight stationary, the values in a convolution window are distributed in rows. So this is equal to spatial_constraint.factors['R'] *spatial_constraint.factors['S'] *spatial_constraint.factors['C'].
- Number of unrolled weights (t): This should be equal to the number of filters passed through the systolic array. Since each active PE interacts with each filter once, this is equal to pe_constraint.factors['K'].

With these changed values of x,y and t the rest of the formulation is same as weight stationary. However, the GB accesses for a tile are different for the number of accesses for weights, inputs and outputs. There are $x \times y, x \times t$ and $y \times t$ accesses for inputs, weights and outputs respectively. These can be reasoned in a similar way as done for weight stationary in Section 5.1.1. The reuse of the input values is the PE is equal to the number of weights that are passed through it which is t or pe_constraint.factors['K'].

5.1.3 Output Stationary

Output stationary is different from weight stationary and input stationary in various ways. It has staggered passing of values as shown in Figure 4.1 and output is kept stationary at PE moving it only when its computation is done. However similar to previous formulations we can define the following:

- *Number of utilized cols (y)*: It is equal to the number of filters that are spatially divided which is spatial_constraint.factors['K'].
- *Number of utilized rows (x)*: Since the convolution window for a output is divided among the PEs, the number of utilized rows will be equal to

the number of output pixels computed that is spatial_constraint.factors['P']* spatial_constraint.factors['Q'].

- *The length of a unrolled convolution window (t)*: Each active PE performs operation on each input value of the unrolled convolution window so this is equal to pe_constraint.factors['R']* pe_constraint.factors['S']* pe_constraint.factors['C'].

With the above definition we can break the whole process in the following parts for counting the number of cycles as shown in Figure 5.5:

- Step 1: The number of cycles before the first element of the convolution window for the last row reaches the systolic array. This is equal to x or the number of utilized rows.
- Step 2: The number of cycles it takes for the last input value in the convolution window to reach the systolic array i.e. *y4* to reach the first col of the systolic array after *s1* has reached the same. This is equal to t or the length of an unrolled convolution window.
- Step 3: The cycles it takes for the last input value in the convolution window to reach the PE at the last active col which is y or the number of utilized cols.
- Step 4: Finally the cycles to move the output data from the bottom-most PE to the GB. This is again equal to x.

The total cycles comes up to be *t+2x+y-3* because of overlapping cycles of the above steps. For GB accesses there are $x \times y$, $x \times t$ and $y \times t$ accesses for outputs, inputs and weights respectively. The reuse of the output values is equal to t or $pe.R \times pe.S \times pe.C$ where pe.Idx denotes pe_constraint.factos[Idx]. But it should be noted that the relation of *x*,*y*, and t with the mapping constraints is different for each stationary type.

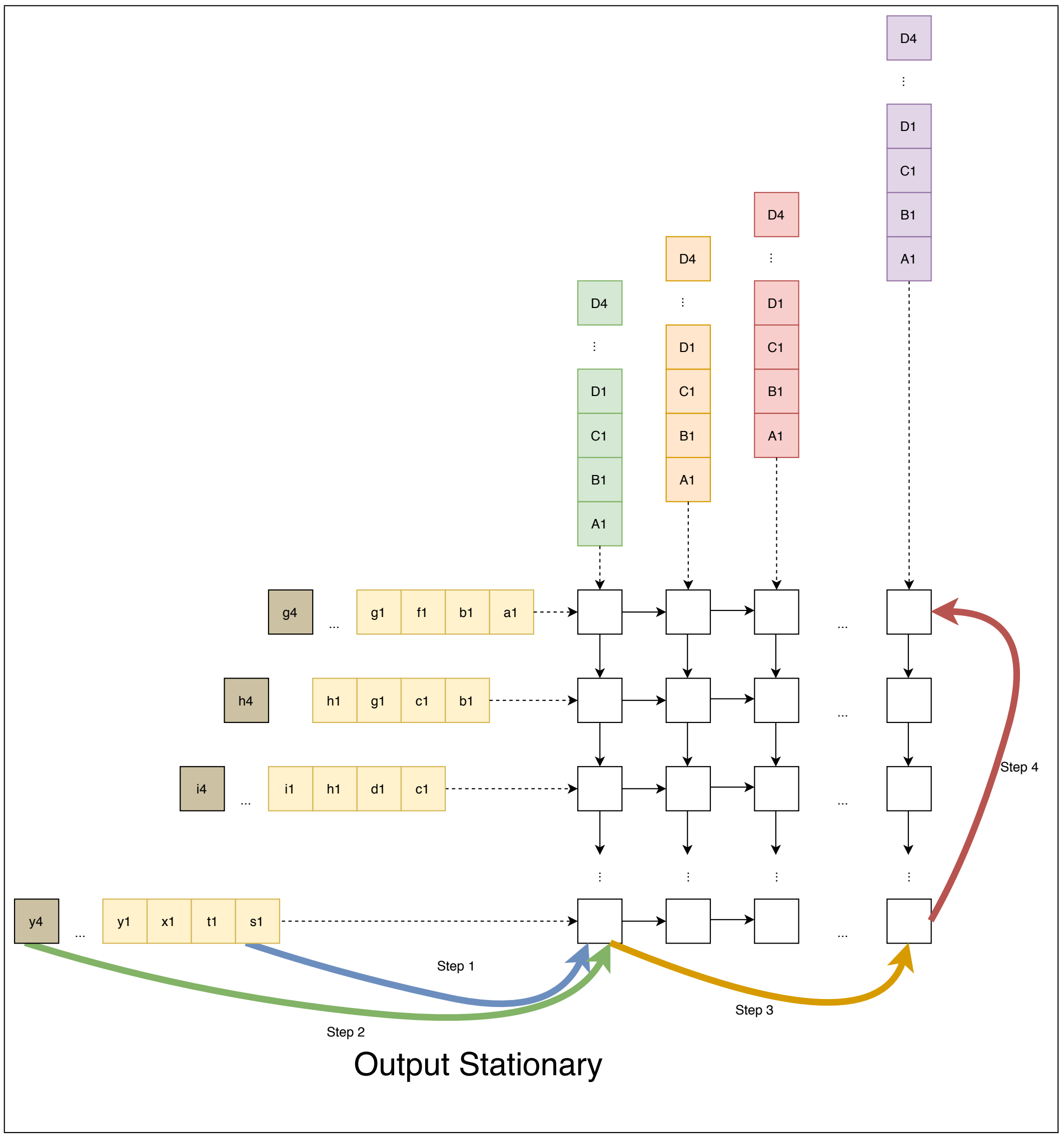

Figure 5.5: Output Stationary

5.1.4 DRAM Accesses

It should be noted that the DRAM accesses for any dataflow depend only on the dram_constraint. Note that Idx/dc.Idx (where dc.Idx is dram_constraint.factors[Idx]) denotes the number of values of Idx which are accessed in one iteration of the nested loops of dram_constraint. For example, if the value of P is 32 in the problem and its factor in dram_constraint is 2 then the 16 (32/2) values of P will be accessed in each iteration. Since Weight and output directly depend on P, Q, R, S, C and K, their accesses can be counted by multiplying the ranges for their respective index variables. That is for weight, it should be ranges of R, S, C, and K, while for output it should be P, Q, and K. However, for input values, the convolution windows size (R and S) should also be included in calculating the width and height.

Chapter 6

Conclusion

In the thesis, Tejas-CNN is presented which is a CNN accelerator simulator. Tejas-CNN is able to simulate an end-to-end neural network on a systolic array based accelerator with any input image. It supports three dataflows namely output, input, and weight stationary which it simulates iteratively cycle by cycle. Since Tejas-CNN is highly modular any new advances in the field can be easily integrated into the framework. By performing real computations Tejas-CNN provides a platform to try out different precision reduction and optimization techniques for enhancing the performance of the system. Furthermore, an analytical model has been described to validate its results and which can be useful in creating a mapping finder.

Bibliography

[1] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition, 1998.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[4] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[5] Hsiang-Tsung Kung. Why systolic architectures?, 1982.

[6] Clement Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision, 2011.

[7] Clement Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks, 2009.

[8] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Memory access optimized routing scheme for deep networks on a mobile coprocessor, 2016.

[9] Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, and Eugenio Culurciello. Memory access optimized routing scheme for deep networks on a mobile coprocessor, 2014.

[10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks.

[11] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[12] Yannan Nellie Wu and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. 2019.

[13] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator, 2019.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[15] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. Tejas: A java based versatile micro-architectural simulator. In *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 47–54. IEEE, 2015.